Ecma/TC39/2015/029 (Rev. 1 - 20 April 2015)
Ecma/GA/2015/033

Final draft Standard ECMA-402

2nd Edition / April 2015

ECMAScript 2015 Internationalization API Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Ecma International
Rue du Rhone 114
CH-1204 Geneva
Tel: +41 22 849 6000
Fax: +41 22 849 6001
Web: http://www.ecma-international.org

Contents

Page

Introduction

The ECMAScript 2015 Internationalization API Specification (ECMA-402 2nd Edition), provides key language-sensitive functionality as a complement to the ECMAScript 2015 Language Specification (ECMA-262 6th Edition or successor). Its functionality has been selected from that of well-established internationalization APIs such as those of the Internationalization Components for Unicode (ICU) library, of the .NET framework, or of the Java platform.

The 1st Edition API was developed by an ad-hoc group established by Ecma TC39 in September 2010 based on a proposal by Nebojša Ćirić and Jungshik Shin.

Internationalization of software is never complete. We expect significant enhancements in future editions of this specification.

Editor, 2nd Edition

Rick Waldron

Contributors

Norbert Lindenberg
Allen Wirfs-Brock
André Bargull
Steven R. Loomis

Editor, 1st Edition

Norbert Lindenberg

Contributors

Eric Albright
Nebojša Ćirić
Peter Constable
Mark Davis
Richard Gillam
Steven Loomis
Mihai Nita
Addison Phillips
Roozbeh Pournader
Jungshik Shin
Shawn Steele
Allen Wirfs-Brock

Feedback provided by Erik Arvidsson, John J. Barton, Zbigniew Braniecki, Marcos Cáceres, Brendan Eich, John Emmons, Gordon P. Hemsley, David Herman, Luke Hoban, Oliver Hunt, Suresh Jayabalan, Yehuda Katz, Mark S. Miller, Andrew Paprocki, Adam Peller, Axel Rauschmayer, Andreas Rossberg, Alex Russell, Markus Scherer, Dmitry Soshnikov, Yusuke Suzuki, John Tamplin, Rick Waldron, Anton Yatsenko, Nicholas Zakas.

This Ecma Standard has been adopted by the General Assembly of <month> <year>.

ECMAScript 2015 Internationalization API Specification

1 Scope

This Standard defines the application programming interface for ECMAScript objects that support programs that need to adapt to the linguistic and cultural conventions used by different human languages and countries.

2 Conformance

A conforming implementation of the ECMAScript 2015 Internationalization API Specification must conform to the ECMAScript 2015 Language Specification (ECMA-262 6th Edition, or successor), and must provide and support all the objects, properties, functions, and program semantics described in this specification.

A conforming implementation of the ECMAScript 2015 Internationalization API Specification is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of the ECMAScript 2015 Internationalization API Specification is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification. A conforming implementation is not permitted to add optional arguments to the functions defined in this specification.

A conforming implementation is permitted to accept additional values, and then have implementation-defined behaviour instead of throwing a **RangeError**, for the following properties of *options* arguments:

- The *options* property localeMatcher in all constructors and supportedLocalesOf methods.
- The *options* properties usage and sensitivity in the Collator constructor.
- The *options* properties style and currencyDisplay in the NumberFormat constructor.
- The *options* properties minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, and maximumSignificantDigits in the NumberFormat constructor, provided that the additional values are interpreted as integer values higher than the specified limits.
- The *options* properties listed in table 3 in the DateTimeFormat constructor.
- The *options* property formatMatcher in the DateTimeFormat constructor.

3 Normative References

The following referenced documents are required for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ECMAScript 2015 Language Specification (ECMA-262 6th Edition, or successor).
http://www.ecma-international.org/publications/standards/Ecma-262.htm

NOTE Throughout this document, the phrase "ES2015, *x*" (where *x* is a sequence of numbers separated by periods) may be used as shorthand for "ECMAScript 2015 Language Specification (ECMA-262 6th Edition, sub clause *x*)".

ISO/IEC 10646:2003: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005 and Amendment 2:2006, Amendment 3:2008, and Amendment 4:2008, plus additional amendments and corrigenda, or successor

http://www.iso.org/iso/catalogue_detail.htm?csnumber=39921
http://www.iso.org/iso/catalogue_detail.htm?csnumber=40755
http://www.iso.org/iso/catalogue_detail.htm?csnumber=41419
http://www.iso.org/iso/catalogue_detail.htm?csnumber=44871
http://www.iso.org/iso/catalogue_detail.htm?csnumber=46311

ISO 4217:2008, Codes for the representation of currencies and funds, or successor
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=46121

IETF BCP 47:

- RFC 5646, Tags for Identifying Languages, or successor
  http://tools.ietf.org/html/rfc5646
- RFC 4647, Matching of Language Tags, or successor
  http://tools.ietf.org/html/rfc4647

IETF RFC 6067, BCP 47 Extension U, or successor
http://tools.ietf.org/html/rfc6067

IANA Time Zone Database
http://www.iana.org/time-zones/

The Unicode Standard, Version 5.1, or successor
http://www.unicode.org/versions/latest

Unicode Technical Standard 35, Unicode Locale Data Markup Language, version 21.0.1 or successor
http://unicode.org/reports/tr35/

4 Overview

This section contains a non-normative overview of the ECMAScript 2015 Internationalization API Specification.

4.1 Internationalization, Localization, and Globalization

Internationalization of software means designing it such that it supports or can be easily adapted to support the needs of users speaking different languages and having different cultural expectations, and enables worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization of software is commonly understood to be the combination of internationalization and localization. Globalization starts at the lowest level by using a text representation that supports all languages in the world, and using standard identifiers to identify languages, countries, time zones, and other relevant parameters. It continues with using a user interface language and data presentation that the user understands, and finally often requires product-specific adaptations to the user's language, culture, and environment.

The ECMAScript 2015 Language Specification lays the foundation by using Unicode for text representation and by providing a few language-sensitive functions, but gives applications little control over the behaviour of these functions. The ECMAScript 2015 Internationalization API Specification builds on this by providing a set of customizable language-sensitive functionality. The API is useful even for applications that themselves are not internationalized, as even applications targeting only one language and one region need to properly support that one language and region. However, the API also enables applications that support multiple languages and regions, even concurrently, as may be needed in server environments.

4.2 API Overview

The ECMAScript 2015 Internationalization API Specification is designed to complement the ECMAScript 2015 Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript 2015 Language Specification (ECMA-262 6th Edition, or successor).

The ECMAScript 2015 Internationalization API Specification provides several key pieces of language-sensitive functionality that are required in most applications: String comparison (collation), number formatting, date and time formatting, and case conversion. While the ECMAScript 2015 Language Specification provides functions for this basic functionality (on Array.prototype: toLocaleString; on String.prototype: localeCompare, toLocaleLowerCase, toLocaleUpperCase; on Number.prototype: toLocaleString; on Date.prototype: toLocaleString, toLocaleDateString, and toLocaleTimeString), it leaves the actual behaviour of these functions largely up to implementations to define. The ECMAScript 2015 Internationalization API Specification provides additional functionality, control over the language and over details of the behaviour to be used, and a more complete specification of required functionality.

Applications can use the API in two ways:

1. Directly, by using the constructors Intl.Collator, Intl.NumberFormat, or Intl.DateTimeFormat to construct an object, specifying a list of preferred languages and options to configure the behaviour of the resulting object. The object then provides a main function (compare or format), which can be called repeatedly. It also provides a resolvedOptions function, which the application can use to find out the exact configuration of the object.

2. Indirectly, by using the functions of the ECMAScript 2015 Language Specification mentioned above. The collation and formatting functions are respecified in this specification to accept the same arguments as the Collator, NumberFormat, and DateTimeFormat constructors and produce the same results as their compare or format methods. The case conversion functions are respecified to accept a list of preferred languages.

The Intl object is used to package all functionality defined in the ECMAScript 2015 Internationalization API Specification to avoid name collisions.

4.3 Implementation Dependencies

Due to the nature of internationalization, the API specification has to leave several details implementation dependent:

- *The set of locales that an implementation supports with adequate localizations:* Linguists estimate the number of human languages to around 6000, and the more widely spoken ones have variations based on regions or other parameters. Even large locale data collections, such as the Common Locale Data Repository, cover only a subset of this large set. Implementations targeting resource-constrained devices may have to further reduce the subset.
- *The exact form of localizations such as format patterns:* In many cases locale-dependent conventions are not standardized, so different forms may exist side by side, or they vary over time. Different internationalization libraries may have implemented different forms, without any of them being actually wrong. In order to allow this API to be implemented on top of existing libraries, such variations have to be permitted.
- *Subsets of Unicode:* Some operations, such as collation, operate on strings that can include characters from the entire Unicode character set. However, both the Unicode standard and the ECMAScript standard allow implementations to limit their functionality to subsets of the Unicode character set. In addition, locale conventions typically don't specify the desired behaviour for the entire Unicode character set, but only for those characters that are relevant for the locale. While the Unicode Collation Algorithm combines a default collation order for the entire Unicode character set with the ability to tailor for local conventions, subsets and tailorings still result in differences in behaviour.

5 Notational Conventions

This standard uses a subset of the notational conventions of the ECMAScript 2015 Language Specification (ECMA-262 6th Edition), as ES2015:

- Object Internal Methods and Internal Slots, as described in ES2015, 6.1.7.2.
- Algorithm conventions, including the use of abstract operations, as described in ES2015, 7.1, 7.2, 7.3.

- Internal Slots, as described in ES2015, 9.1.
- The List and Record Specification Type, as described in ES2015, 6.2.1.

NOTE As described in the ECMAScript Language Specification, algorithms are used to precisely specify the required semantics of ECMAScript constructs, but are not intended to imply the use of any specific implementation technique. Internal slots are used to define the semantics of object values, but are not part of the API. They are defined purely for expository purposes. An implementation of the API must behave as if it produced and operated upon internal slots in the manner described here.

As an extension to the Record Specification Type, the notation "[[<*name*>]]" denotes a field whose name is given by the variable *name*, which must have a String value. For example, if a variable *s* has the value `"a"`, then [[<*s*>]] denotes the field [[a]].

For ECMAScript objects, this standard may use variable-named internal slots: The notation "[[<*name*>]]" denotes an internal slot whose name is given by the variable *name*, which must have a String value. For example, if a variable *s* has the value `"a"`, then [[<*s*>]] denotes the [[a]] internal slot.

5.1 Well-Known Intrinsic Objects

The following table extends the Well-Known Intrinsic Objects table defined in ES2015, 6.1.7.4.

Table 7 — Well-known Intrinsic Objects (Extensions)

Intrinsic Name	Global Name	ECMAScript Language Association
%Date_now%	"Date.now"	The initial value of the `"now"` data property of the intrinsic %Date% (ES2015, 20.3.3.1)
%Intl%	"Intl"	The Intl object (8).
%Collator%	"Intl.Collator"	The Intl.Collator constructor (10.1)
%CollatorPrototype%	"Intl.Collator.prototype"	The initial value of the "`prototype`" data property of the intrinsic %Collator% (10.2.1).
%NumberFormat%	"Intl.NumberFormat"	The Intl.NumberFormat constructor (11.1)
%NumberFormatPrototype%	"Intl.NumberFormat.prototype"	The initial value of the "`prototype`" data property of the intrinsic %NumberFormat% (11.2.1).
%DateTimeFormat%	"Intl.DateTimeFormat"	The Intl.DateTimeFormat constructor (12.1).
%DateTimeFormatPrototype%	"Intl.DateTimeFormat.prototype"	The initial value of the "`prototype`" data property of the intrinsic %DateTimeFormat% (11.2.1).
%StringProto_split%	"String.prototype.split"	The initial value of the `"split"` data property of the

		intrinsic %StringPrototype% (ES2015, 21.1.3.17)
%StringProto_indexOf%	"String.prototype.indexOf"	The initial value of the `"indexOf"` data property of the intrinsic %StringPrototype% (ES2015, 21.1.3.8)
%StringProto_includes%	"String.prototype.includes"	The initial value of the `"includes"` data property of the intrinsic %StringPrototype% (ES2015, 21.1.3.7)
%ArrayProto_indexOf%	"Array.prototype.indexOf"	The initial value of the `"indexOf"` data property of the intrinsic %StringPrototype% (ES2015, 22.1.3.11)

6 Identification of Locales, Currencies, and Time Zones

This clause describes the String values used in the ECMAScript 2015 Internationalization API Specification to identify locales, currencies, and time zones.

6.1 Case Sensitivity and Case Mapping

The String values used to identify locales, currencies, and time zones are interpreted in a case-insensitive manner, treating the Unicode Basic Latin characters "A" to "Z" (U+0041 to U+005A) as equivalent to the corresponding Basic Latin characters "a" to "z" (U+0061 to U+007A). No other case folding equivalences are applied. When mapping to upper case, a mapping shall be used that maps characters in the range "a" to "z" (U+0061 to U+007A) to the corresponding characters in the range "A" to "Z" (U+0041 to U+005A) and maps no other characters to the latter range.

EXAMPLES "ß" (U+00DF) must not match or be mapped to "SS" (U+0053, U+0053). "ı" (U+0131) must not match or be mapped to "I" (U+0049).

6.2 Language Tags

The ECMAScript 2015 Internationalization API Specification identifies locales using language tags as defined by IETF BCP 47 (RFCs 5646 and 4647 or their successors), which may include extensions such as those registered through RFC 6067. Their canonical form is specified in RFC 5646 section 4.5 or its successor.

BCP 47 language tags that meet those validity criteria of RFC 5646 section 2.2.9 that can be verified without reference to the IANA Language Subtag Registry are considered structurally valid. All structurally valid language tags are valid for use with the APIs defined by this standard. However, the set of locales and thus language tags that an implementation supports with adequate localizations is implementation dependent. The constructors Collator, NumberFormat, and DateTimeFormat map the language tags used in requests to locales supported by their respective implementations.

6.2.1 Unicode Locale Extension Sequences

This standard uses the term "Unicode locale extension sequence" for any substring of a language tag that is not part of a private use subtag sequence, starts with a separator "-" and the singleton "u", and includes the maximum sequence of following non-singleton subtags and their preceding "-" separators.

6.2.2 IsStructurallyValidLanguageTag (locale)

The IsStructurallyValidLanguageTag abstract operation verifies that the *locale* argument (which must be a String value)

- represents a well-formed BCP 47 language tag as specified in RFC 5646 section 2.1, or successor,
- does not include duplicate variant subtags, and
- does not include duplicate singleton subtags.

The abstract operation returns true if *locale* can be generated from the ABNF grammar in section 2.1 of the RFC, starting with Language-Tag, and does not contain duplicate variant or singleton subtags (other than as a private use subtag). It returns false otherwise. Terminal value characters in the grammar are interpreted as the Unicode equivalents of the ASCII octet values given.

6.2.3 CanonicalizeLanguageTag (locale)

The CanonicalizeLanguageTag abstract operation returns the canonical and case-regularized form of the *locale* argument (which must be a String value that is a structurally valid BCP 47 language tag as verified by the IsStructurallyValidLanguageTag abstract operation). It takes the steps specified in RFC 5646 section 4.5, or successor, to bring the language tag into canonical form, and to regularize the case of the subtags, but does not take the steps to bring a language tag into "extlang form" and to reorder variant subtags.

The specifications for extensions to BCP 47 language tags, such as RFC 6067, may include canonicalization rules for the extension subtag sequences they define that go beyond the canonicalization rules of RFC 5646 section 4.5. Implementations are allowed, but not required, to apply these additional rules.

6.2.4 DefaultLocale ()

The DefaultLocale abstract operation returns a String value representing the structurally valid (6.2.2) and canonicalized (6.2.3) BCP 47 language tag for the host environment's current locale.

6.3 Currency Codes

The ECMAScript 2015 Internationalization API Specification identifies currencies using 3-letter currency codes as defined by ISO 4217. Their canonical form is upper case.

All well-formed 3-letter ISO 4217 currency codes are allowed. However, the set of combinations of currency code and language tag for which localized currency symbols are available is implementation dependent. Where a localized currency symbol is not available, the ISO 4217 currency code is used for formatting.

6.3.1 IsWellFormedCurrencyCode (currency)

The IsWellFormedCurrencyCode abstract operation verifies that the *currency* argument (which must be a String value) represents a well-formed 3-letter ISO currency code. The following steps are taken:

1. Let *normalized* be the result of mapping *currency* to upper case as described in 6.1.
2. If the string length of *normalized* is not 3, return **false**.
3. If *normalized* contains any character that is not in the range "A" to "Z" (U+0041 to U+005A), return **false**.
4. Return **true**.

6.4 Time Zone Names

The ECMAScript 2015 Internationalization API Specification identifies time zones using the Zone and Link names of the IANA Time Zone Database. Their canonical form is the corresponding Zone name in the casing used in the IANA Time Zone Database.

All registered Zone and Link names are allowed. Implementations must recognize all such names, and use best available current and historical information about their offsets from UTC and their daylight saving time rules in calculations. However, the set of combinations of time zone name and language tag for which localized time zone names are available is implementation dependent.

6.4.1 IsValidTimeZoneName (timeZone)

The IsValidTimeZoneName abstract operation verifies that the *timeZone* argument (which must be a String value) represents a valid Zone or Link name of the IANA Time Zone Database.

The abstract operation returns true if *timeZone*, converted to upper case as described in 6.1, is equal to one of the Zone or Link names of the IANA Time Zone Database, converted to upper case as described in 6.1. It returns false otherwise.

6.4.2 CanonicalizeTimeZoneName (timeZone)

The CanonicalizeTimeZoneName abstract operation returns the canonical and case-regularized form of the *timeZone* argument (which must be a String value that is a valid time zone name as verified by the IsValidTimeZoneName abstract operation). The following steps are taken:

1. Let *ianaTimeZone* be the Zone or Link name of the IANA Time Zone Database such that *timeZone*, converted to upper case as described in 6.1, is equal to *ianaTimeZone*, converted to upper case as described in 6.1.
2. If *ianaTimeZone* is a Link name, then let *ianaTimeZone* be the corresponding Zone name as specified in the "backward" file of the IANA Time Zone Database.
3. If *ianaTimeZone* is `"Etc/UTC"` or `"Etc/GMT"`, then return `"UTC"`.
4. Return *ianaTimeZone*.

The Intl.DateTimeFormat constructor allows this time zone name; if the time zone is not specified, the host environment's current time zone is used. Implementations shall support UTC and the host environment's current time zone (if different from UTC) in formatting.

6.4.3 DefaultTimeZone ()

The DefaultTimeZone abstract operation returns a String value representing the valid (6.4.1) and canonicalized (6.4.2) time zone name for the host environment's current time zone.

7 Requirements for Standard Built-in ECMAScript Objects

Unless specified otherwise in this document, the objects, functions, and constructors described in this standard are subject to the generic requirements and restrictions specified for standard built-in ECMAScript objects in the ECMAScript 2015 Language Specification, 6th edition, clause 17, or successor.

8 The Intl Object

The Intl object is a single ordinary object.

The value of the [[Prototype]] internal slot of the Intl object is the intrinsic object `%ObjectPrototype%`.

The Intl object is not a function object. It does not have a [[Construct]] internal method; it is not possible to use the Intl object as a constructor with the `new` operator. The Intl object does not have a [[Call]] internal method; it is not possible to invoke the Intl object as a function.

8.1 Properties of the Intl Object

The value of each of the standard built-in properties of the Intl object is a constructor. The behaviour of these constructors is specified in the following clauses: Collator (10), NumberFormat (11), and DateTimeFormat (12).

9 Locale and Parameter Negotiation

The constructors for the objects providing locale sensitive services, Collator, NumberFormat, and DateTimeFormat, use a common pattern to negotiate the requests represented by the locales and options arguments against the actual capabilities of their implementations. The common behaviour is described here in terms of internal slots describing the capabilities and of abstract operations using these internal slots.

9.1 Internal slots of Service Constructors

The constructors Intl.Collator, Intl.NumberFormat, and Intl.DateTimeFormat have the following internal slots:

- [[availableLocales]] is a List that contains structurally valid (6.2.2) and canonicalized (6.2.3) BCP 47 language tags identifying the locales for which the implementation provides the functionality of the constructed objects. Language tags on the list must not have a Unicode locale extension sequence. The list must include the value returned by the DefaultLocale abstract operation (6.2.4), and must not include duplicates. Implementations must include in [[availableLocales]] locales that can serve as fallbacks in the algorithm used to resolve locales (see 9.2.5). For example, implementations that provide a "de-DE" locale must include a "de" locale that can serve as a fallback for requests such as "de-AT" and "de-CH". For locales that in current usage would include a script subtag (such as Chinese locales), old-style language tags without script subtags must be included such that, for example, requests for "zh-TW" and "zh-HK" lead to output in traditional Chinese rather than the default simplified Chinese. The ordering of the locales within [[availableLocales]] is irrelevant.
- [[relevantExtensionKeys]] is an array of keys of the language tag extensions defined in Unicode Technical Standard 35 that are relevant for the functionality of the constructed objects.
- [[sortLocaleData]] and [[searchLocaleData]] (for Intl.Collator) and [[localeData]] (for Intl.NumberFormat and Intl.DateTimeFormat) are objects that have properties for each locale contained in [[availableLocales]]. The value of each of these properties must be an object that has properties for each key contained in [[relevantExtensionKeys]]. The value of each of these properties must be a non-empty array of those values defined in Unicode Technical Standard 35 for the given key that are supported by the implementation for the given locale, with the first element providing the default value.

EXAMPLE An implementation of DateTimeFormat might include the language tag "th" in its [[availableLocales]] internal slot, and must (according to 12.2.3) include the key "ca" in its [[relevantExtensionKeys]] internal slot. For Thai, the "buddhist" calendar is usually the default, but an implementation might also support the calendars "gregory", "chinese", and "islamicc" for the locale "th". The [[localeData]] internal slot would therefore at least include {"th": {ca: ["buddhist", "gregory", "chinese", "islamicc"]}}.

9.2 Abstract Operations

Where the following abstract operations take an *availableLocales* argument, it must be an [[availableLocales]] List as specified in 9.1.

9.2.1 CanonicalizeLocaleList (locales)

The abstract operation CanonicalizeLocaleList takes the following steps:

1. If *locales* is **undefined**, then
    a. Return a new empty List.
2. Let *seen* be an empty List.
3. If Type(*locales*) is String, then
    a. Let *aLocales* be CreateArrayFromList(«*locales*»).
4. Let *O* be ToObject(*aLocales*).
5. ReturnIfAbrupt(*O*).
6. Let *len* be ToLength(Get(*O*, "`length`")).
7. Let *k* be 0.
8. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be HasProperty(*O, Pk*).
    c. ReturnIfAbrupt(*kPresent*).

   d. If *kPresent* is **true**, then
      i. Let *kValue* be Get(*O*, *Pk*).
      ii. ReturnIfAbrupt(*kValue*).
      iii. If Type(*kValue*) is not String or Object, throw a **TypeError** exception.
      iv. Let *tag* be ToString(*kValue*).
      v. ReturnIfAbrupt(*tag*).
      vi. If IsStructurallyValidLanguageTag(*tag*) is **false**, throw a **RangeError** exception.
      vii. Let *canonicalizedTag* be CanonicalizeLanguageTag(*tag*).
      viii. If *canonicalizedTag* is not an element of *seen*, append *canonicalizedTag* as the last element of *seen*.
   e. Increase *k* by 1.
9. Return *seen*.

NOTE Non-normative summary: The abstract operation interprets the *locales* argument as an array and copies its elements into a List, validating the elements as structurally valid language tags and canonicalizing them, and omitting duplicates.

NOTE Requiring *kValue* to be a String or Object means that the Number value **NaN** will not be interpreted as the language tag `"nan"`, which stands for Min Nan Chinese.

9.2.2 BestAvailableLocale (availableLocales, locale)

The BestAvailableLocale abstract operation compares the provided argument *locale*, which must be a String value with a structurally valid and canonicalized BCP 47 language tag, against the locales in *availableLocales* and returns either the longest non-empty prefix of *locale* that is an element of *availableLocales*, or **undefined** if there is no such element. It uses the fallback mechanism of RFC 4647, section 3.4. The following steps are taken:

1. Let *candidate* be *locale*.
2. Repeat
   a. If *availableLocales* contains an element equal to *candidate*, then return *candidate*.
   b. Let *pos* be the character index of the last occurrence of `"-"` (U+002D) within *candidate*. If that character does not occur, return **undefined**.
   c. If *pos* ≥ 2 and the character `"-"` occurs at index *pos*-2 of *candidate*, then decrease *pos* by 2.
   d. Let *candidate* be the substring of *candidate* from position 0, inclusive, to position *pos*, exclusive.

9.2.3 LookupMatcher (availableLocales, requestedLocales)

The LookupMatcher abstract operation compares *requestedLocales*, which must be a List as returned by CanonicalizeLocaleList, against the locales in *availableLocales* and determines the best available language to meet the request. The following steps are taken:

1. Let *k* be 0.
2. Let *rLocales* be CreateArrayFromList(*requestedLocales*).
3. Let *len* be ToLength(Get(*rLocales*, `"length"`)).
4. Let *availableLocale* be **undefined**.
5. Repeat while *k* < *len* and *availableLocale* is **undefined**:
   a. Let *Pk* be ToString(*k*).
   b. Let *locale* be Get(*rLocales*, *Pk*) .
   c. ReturnIfAbrupt(*locale*)
   d. Let *noExtensionsLocale* be the String value that is *locale* with all Unicode locale extension sequences removed.
   e. Let *availableLocale* be BestAvailableLocale(*availableLocales*, *noExtensionsLocale*).
   f. Increase *k* by 1.
6. Let *result* be a new Record.
7. If *availableLocale* is not **undefined**, then
   a. Set *result*.[[locale]] to *availableLocale*.
   b. If *locale* and *noExtensionsLocale* are not the same String value, then

i. Let *extension* be the String value consisting of the first substring of *locale* that is a Unicode locale extension sequence.
ii. Let *extensionIndex* be the character position of the initial `"-"` of the first Unicode locale extension sequence within *locale*.
iii. Set *result*.[[extension]] to *extension*.
iv. Set *result*.[[extensionIndex]] to *extensionIndex*.

8. Else
    a. Let *defLocale* be DefaultLocale().
    b. Set *result*.[[locale]] to *defLocale*.
9. Return *result*.

NOTE The algorithm is based on the Lookup algorithm described in RFC 4647 section 3.4, but options specified through Unicode locale extension sequences are ignored in the lookup. Information about such subsequences is returned separately. The abstract operation returns a record with a [[locale]] field, whose value is the language tag of the selected locale, which must be an element of *availableLocales*. If the language tag of the request locale that led to the selected locale contained a Unicode locale extension sequence, then the returned record also contains an [[extension]] field whose value is the first Unicode locale extension sequence, and an [[extensionIndex]] field whose value is the index of the first Unicode locale extension sequence within the request locale language tag.

9.2.4 BestFitMatcher (availableLocales, requestedLocales)

The BestFitMatcher abstract operation compares *requestedLocales*, which must be a List as returned by CanonicalizeLocaleList, against the locales in *availableLocales* and determines the best available language to meet the request. The algorithm is implementation dependent, but should produce results that a typical user of the requested locales would perceive as at least as good as those produced by the LookupMatcher abstract operation. Options specified through Unicode locale extension sequences must be ignored by the algorithm. Information about such subsequences is returned separately. The abstract operation returns a record with a [[locale]] field, whose value is the language tag of the selected locale, which must be an element of *availableLocales*. If the language tag of the request locale that led to the selected locale contained a Unicode locale extension sequence, then the returned record also contains an [[extension]] field whose value is the first Unicode locale extension sequence, and an [[extensionIndex]] field whose value is the index of the first Unicode locale extension sequence within the request locale language tag.

9.2.5 ResolveLocale (availableLocales, requestedLocales, options, relevantExtensionKeys, localeData)

The ResolveLocale abstract operation compares a BCP 47 language priority list *requestedLocales* against the locales in *availableLocales* and determines the best available language to meet the request. *availableLocales* and *requestedLocales* must be provided as List values, *options* as a Record.

The following steps are taken:

1. Let *matcher* be the value of *options*.[[localeMatcher]].
2. If *matcher* is **`"lookup"`**, then
    a. Let *MatcherOperation* be the abstract operation LookupMatcher.
3. Else
    a. Let *MatcherOperation* be the abstract operation BestFitMatcher.
4. Let r be *MatcherOperation*(*availableLocales*, *requestedLocales*).
5. Let *foundLocale* be the value of *r*.[[locale]].
6. If *r* has an [[extension]] field, then
    a. Let *extension* be the value of *r*.[[extension]].
    b. Let *extensionIndex* be the value of *r*.[[extensionIndex]].
    c. Let *extensionSubtags* be Call(**`%StringProto_split%`**, *extension,* «`"-"`») .
    d. Let *extensionSubtagsLength* be Get(CreateArrayFromList(*extensionSubtags*)*,* **`"length"`**).
7. Let *result* be a new Record.
8. Set *result*.[[dataLocale]] to *foundLocale*.
9. Let *supportedExtension* be **`"-u"`**.
10. Let *k* be 0.

11. Let *rExtensionKeys* be ToObject(CreateArrayFromList(*relevantExtensionKeys*)).
12. ReturnIfAbrupt(*rExtensionKeys*).
13. Let *len* be ToLength(Get(*rExtensionKeys,* **`"length"`**)).
14. Repeat while *k* < *len*
    a. Let *key* be Get(*rExtensionKeys* , ToString(*k*)).
    b. ReturnIfAbrupt(*key*).
    c. Let *foundLocaleData* be Get(*localeData, foundLocale*).
    d. ReturnIfAbrupt(*foundLocaleData*).
    e. Let *keyLocaleData* be ToObject(Get(*foundLocaleData, key*)).
    f. ReturnIfAbrupt(*keyLocaleData*).
    g. Let *value* be ToString(Get(*keyLocaleData*, **`"0"`**)).
    h. ReturnIfAbrupt(*value*).
    i. Let *supportedExtensionAddition* be `""`.
    j. If *extensionSubtags* is not **undefined**, then
        i. Let *keyPos* be Call(**`%ArrayProto_indexOf%`**, *extensionSubtags,* «*key*») .
        ii. If *keyPos* ≠ -1, then
            1. If *keyPos* + 1 < *extensionSubtagsLength* and the length of the result of Get(*extensionSubtags*, ToString(*keyPos* +1)) is greater than 2, then
                a. Let *requestedValue* be Get(*extensionSubtags*, ToString(*keyPos* +1)).
                b. If the result of Call(**`%StringProto_includes%,`** *keyLocaleData,* «*requestedValue*») is **true**, then
                    i. Let *value* be *requestedValue*.
                    ii. Let *supportedExtensionAddition* be the concatenation of **`"-"`**, *key*, **`"-"`**, and *value*.
            2. Else, if the result of Call(**`%StringProto_includes%`***, keyLocaleData,* «**`"true"`**») is **true**, then
                    i. Let *value* be **`"true"`**.
    k. If *options* has a field [[<*key*>]], then
        i. Let *optionsValue* be the value of ToString(*options*.[[<*key*>]]).
        ii. ReturnIfAbrupt(*optionsValue*).
        iii. If the result of Call(**`%StringProto_includes%,`** *keyLocaleData,* «*optionsValue*») is **true**, then
            1. If *optionsValue* is not equal to *value*, then
                a. Let *value* be *optionsValue*.
                b. Let *supportedExtensionAddition* be `""`.
    l. Set *result*.[[<*key*>]] to *value*.
    m. Append *supportedExtensionAddition* to *supportedExtension*.
    n. Increase *k* by 1.
15. If the number of elements in *supportedExtension* is greater than 2, then
    a. Let *preExtension* be the substring of *foundLocale* from position 0, inclusive, to position *extensionIndex*, exclusive.
    b. Let *postExtension* be the substring of *foundLocale* from position *extensionIndex* to the end of the string.
    c. Let *foundLocale* be the concatenation of *preExtension*, *supportedExtension*, and *postExtension*.
16. Set *result*.[[locale]] to *foundLocale*.
17. Return *result*.

NOTE Non-normative summary: Two algorithms are available to match the locales: the Lookup algorithm described in RFC 4647 section 3.4, and an implementation dependent best-fit algorithm. Independent of the locale matching algorithm, options specified through Unicode locale extension sequences are negotiated separately, taking the caller's relevant extension keys and locale data as well as client-provided options into consideration. The abstract operation returns a record with a [[locale]] field whose value is the language tag of the selected locale, and fields for each key in *relevantExtensionKeys* providing the selected value for that key.

9.2.6 LookupSupportedLocales (availableLocales, requestedLocales)

The LookupSupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale when using the BCP 47 Lookup algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The following steps are taken:

1. Let *rLocales* be CreateArrayFromList(*requestedLocales*).
2. Let *len* be ToLength(Get(*rLocales*, **`"length"`**)).

3. Let *subset* be an empty List.
4. Let *k* be 0.
5. Repeat while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *locale* be Get(*rLocales*, *Pk*).
    c. ReturnIfAbrupt(*locale*).
    d. Let *noExtensionsLocale* be the String value that is *locale* with all Unicode locale extension sequences removed.
    e. Let *availableLocale* be BestAvailableLocale(*availableLocales, noExtensionsLocale*).
    f. If *availableLocale* is not **undefined**, then append *locale* to the end of *subset*.
    g. Increment *k* by 1.
6. Return *subset*.

9.2.7 BestFitSupportedLocales (availableLocales, requestedLocales)

The BestFitSupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale when using the Best Fit Matcher algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The steps taken are implementation dependent.

9.2.8 SupportedLocales (availableLocales, requestedLocales, options)

The SupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale. Two algorithms are available to match the locales: the Lookup algorithm described in RFC 4647 section 3.4, and an implementation dependent best-fit algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The following steps are taken:

1. If *options* is not **undefined**, then
    a. Let *matcher* be GetOption(*options*, "localeMatcher", "string", « "lookup", "best fit" », "best fit").
    b. ReturnIfAbrupt(*matcher*).
2. Else, let *matcher* be `"best fit"`.
3. If *matcher* is `"best fit"`,
    a. Let *MatcherOperation* be the abstract operation BestFitSupportedLocales.
4. Else
    a. Let *MatcherOperation* be the abstract operation LookupSupportedLocales.
5. Let supportedLocales be *MatcherOperation*(*availableLocales*, *requestedLocales*).
6. Let *subset* be CreateArrayFromList(*supportedLocales*).
7. Let *keys* be *subset*.[[OwnPropertyKeys]]().
8. Repeat for each element *P* of *keys* in List order,
    a. Let *desc* be PropertyDescriptor { [[Configurable]]: **false**, [[Writable]]: **false** }.
    b. Let *status* be DefinePropertyOrThrow(*subset, P, desc*).
    c. Assert: *status* is not abrupt completion.
9. Return *subset*.

9.2.9 GetOption (options, property, type, values, fallback)

The abstract operation GetOption extracts the value of the property named *property* from the provided *options* object, converts it to the required *type*, checks whether it is one of a List of allowed *values*, and fills in a *fallback* value if necessary.

1. Let *opts* be ToObject(*options*).
2. ReturnIfAbrupt(*opts*).
3. Let *value* be Get(*opts, property*).
4. ReturnIfAbrupt(*value*).
5. If *value* is not **undefined**, then
    a. Assert: *type* is `"boolean"` or `"string"`.
    b. If *type* is `"boolean"`, then
        i. Let *value* be ToBoolean(*value*).

c. If *type* is **"string"**, then
   i. Let *value* be ToString(*value*).
   ii. ReturnIfAbrupt(*value*).
d. If *values* is not **undefined**, then
   i. If *values* does not contain an element equal to *value*, throw a **RangeError** exception.
e. Return *value*.
6. Else return *fallback*.

9.2.10 GetNumberOption (options, property, minimum, maximum, fallback)

The abstract operation GetNumberOption extracts a property value from the provided options object, converts it to a Number value, checks whether it is in the allowed range, and fills in a fallback value if necessary.

1. Let *opts* be ToObject(*options*).
2. ReturnIfAbrupt(*opts*).
3. Let *value* be Get(*opts, property*).
4. ReturnIfAbrupt(*value*).
5. If *value* is not **undefined**, then
   a. Let *value* be ToNumber(*value*).
   b. ReturnIfAbrupt(*value*).
   a. If *value* is **NaN** or less than *minimum* or greater than *maximum*, throw a **RangeError** exception.
   c. Return floor(*value*).
6. Else return *fallback*.

10 Collator Objects

10.1 The Intl.Collator Constructor

The Intl.Collator constructor is a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

10.1.1 InitializeCollator (collator, locales, options)

The abstract operation InitializeCollator accepts the arguments *collator* (which must be an object), *locales*, and *options*. It initializes *collator* as a **Collator** object.

Several steps in the algorithm use values from the following table, which associates Unicode locale extension keys, property names, types, and allowable values:

Table 1 – Collator options settable through both extension keys and options properties

Key	Property	Type	Values
kn	numeric	"boolean"	
kf	caseFirst	"string"	"upper", "lower", "false"

The following steps are taken:

1. If *collator* has an [[initializedIntlObject]] internal slot with value **true**, throw a **TypeError** exception.
2. Set *collator*.[[initializedIntlObject]] to **true**.
3. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
4. ReturnIfAbrupt(*requestedLocales*).
5. If *options* is **undefined**, then
   a. Let *options* be ObjectCreate(**%ObjectPrototype%**).
6. Else
   a. Let *options* be ToObject(*options*).
   b. ReturnIfAbrupt(*options*).

7. Let *u* be GetOption(*options*, `"usage"`, `"string"`, «`"sort"`, `"search"`», `"sort"`).
8. ReturnIfAbrupt(*u*).
9. Set *collator*.[[usage]] to *u*.
10. If *u* is `"sort"`, then
    a. Let *localeData* be the value of `%Collator%`.[[sortLocaleData]];
11. Else
    a. Let *localeData* be the value of `%Collator%`.[[searchLocaleData]].
12. Let *opt* be a new Record.
13. Let *matcher* be GetOption(*options*, `"localeMatcher"`, `"string"`, «`"lookup"`, `"best fit"`», `"best fit"`).
14. ReturnIfAbrupt(*matcher*).
15. Set *opt*.[[localeMatcher]] to *matcher*.
16. For each row in Table 1, except the header row, do:
    a. Let *key* be the name given in the Key column of the row.
    b. Let *prop* be the name given in the Property column of the row.
    c. Let *type* be the string given in the Type column of the row.
    d. Let *list* be a List containing the Strings given in the Values column of the row, or **undefined** if no strings are given.
    e. Let *value* be GetOption(*options*, *prop*, *type*, *list*, **undefined**).
    f. ReturnIfAbrupt(*value*).
    g. If the string given in the Type column of the row is `"boolean"` and *value* is not **undefined**, then
        i. Let *value* be ToString(*value*).
        ii. ReturnIfAbrupt(*value*).
    h. Set *opt*.[[<*key*>]] to *value*.
17. Let *relevantExtensionKeys* be the value of `%Collator%`.[[relevantExtensionKeys]].
18. Let *r* be ResolveLocale(`%Collator%`.[[availableLocales]], *requestedLocales*, *opt*, *relevantExtensionKeys*, *localeData*).
19. Set *collator*.[[locale]] to the value of *r*.[[locale]].
20. Let *k* be 0.
21. Let *lenValue* be Get(*relevantExtensionKeys,* `"length")`.
22. Let *len* be ToLength(*lenValue*).
23. Repeat while *k* < *len*:
    a. Let *Pk* be ToString(*k*).
    b. Let *key* be Get(*relevantExtensionKeys, Pk*).
    c. ReturnIfAbrupt(*key*).
    d. If *key* is `"co"`, then
        i. Let *property* be `"collation"`.
        ii. Let *value* be the value of *r*.[[co]].
        iii. If *value* is **null**, let *value* be `"default"`.
    e. Else use the row of Table 1 that contains the value of *key* in the Key column:
        i. Let *property* be the name given in the Property column of the row.
        ii. Let *value* be the value of *r*.[[<*key*>]].
        iii. If the name given in the Type column of the row is `"boolean"`, let *value* be the result of comparing *value* with `"true"`.
    f. Set *collator*.[[<*property*>]] to *value*.
    g. Increase *k* by 1.
24. Let *s* be GetOption(*options*, `"sensitivity"`, `"string"`, «`"base"`, `"accent"`, `"case"`, `"variant"`», , **undefined**).
25. ReturnIfAbrupt(*s*).
26. If *s* is **undefined**, then
    a. If *u* is `"sort"`, then let s be `"variant"`.
    b. Else
        i. Let *dataLocale* be the value of *r*.[[dataLocale]].
        ii. Let *dataLocaleData* be Get(*localeData, dataLocale*).
        iii. Let *s* be Get(*dataLocaleData*, `"sensitivity")`.
27. Set *collator*.[[sensitivity]] to *s*.
28. Let *ip* be GetOption(*options*, `"ignorePunctuation"`, `"boolean"`, **undefined**, **false**).
29. ReturnIfAbrupt(*ip*).
30. Set *collator*.[[ignorePunctuation]] to *ip*.
31. Set *collator*.[[boundCompare]] to **undefined**.

32. Set *collator*.[[initializedCollator]] to **true**.
33. Return *collator*.

10.1.2 Intl.Collator([locales [, options]])

When the **`Intl.Collator`** function is called with optional arguments *locales* and *options* the following steps are taken:

1. If NewTarget is **undefined,** let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *collator* be OrdinaryCreateFromConstructor(*newTarget*, **`%CollatorPrototype%`**).
3. ReturnIfAbrupt(*collator*).
4. Return InitializeCollator(*collator, locales, options*).

10.2 Properties of the Intl.Collator Constructor

Besides the internal slots and the **`length`** property (whose value is 0), the Intl.Collator constructor has the following properties:

10.2.1 Intl.Collator.prototype

The value of **`Intl.Collator.prototype`** is **`%CollatorPrototype%`**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

10.2.2 Intl.Collator.supportedLocalesOf (locales [, options])

When the **`supportedLocalesOf`** method of Intl.Collator is called, the following steps are taken:

1. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
2. ReturnIfAbrupt(*requestedLocales*).
3. Return SupportedLocales(**`%Collator%`**.[[availableLocales]], *requestedLocales*, *options*).

The value of the **`length`** property of the **`supportedLocalesOf`** method is **`1`**.

10.2.3 Internal slots

The value of the [[availableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[relevantExtensionKeys]] internal slot is an array that must include the element **`"co"`**, may include any or all of the elements **`"kn"`** and **`"kf"`**, and must not include any other elements.

NOTE Unicode Technical Standard 35 describes ten locale extension keys that are relevant to collation: "co" for collator usage and specializations, "ka" for alternate handling, "kb" for backward second level weight, "kc" for case level, "kn" for numeric, "kh" for hiragana quaternary, "kk" for normalization, "kf" for case first, "kr" for reordering, "ks" for collation strength, and "vt" for variable top. Collator, however, requires that the usage is specified through the usage property of the options object, alternate handling through the ignorePunctuation property of the options object, and case level and the strength through the sensitivity property of the options object. The "co" key in the language tag is supported only for collator specializations, and the keys "kb", "kh", "kk", "kr", and "vt" are not allowed in this version of the Internationalization API. Support for the remaining keys is implementation dependent.

The values of the [[sortLocaleData]] and [[searchLocaleData]] internal slots are implementation defined within the constraints described in 9.1 and the following additional constraints:

- The first element of [[sortLocaleData]][locale].co and [[searchLocaleData]][locale].co must be null for all locale values.
- The values `"standard"` and `"search"` must not be used as elements in any [[sortLocaleData]][locale].co and [[searchLocaleData]][locale].co array.
- [[searchLocaleData]][locale] must have a sensitivity property with a String value equal to `"base"`, `"accent"`, `"case"`, or `"variant"` for all locale values.

10.3 Properties of the Intl.Collator Prototype Object

The Intl.Collator prototype object is itself an Intl.Collator instance as specified in 10.4, whose internal slots are set as if it had been constructed by the expression Construct(`%Collator%`)

In the following descriptions of functions that are properties or [[Get]] attributes of properties of `%CollatorPrototype%`, the phrase "this Collator object" refers to the object that is the **this** value for the invocation of the function; a **TypeError** exception is thrown if the this value is not an object or an object that does not have an [[initializedCollator]] internal slot with value **true**.

10.3.1 Intl.Collator.prototype.constructor

The value of `Intl.Collator.prototype.constructor` is `%Collator%`.

10.3.2 Intl.Collator.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

10.3.3 Intl.Collator.prototype.compare

This named accessor property returns a function that compares two strings according to the sort order of this Collator object.

The value of the [[Get]] attribute is a function that takes the following steps:

1. Let *collator* be this Collator object.
2. If *collator*.[[boundCompare]] is **undefined**, then
    a. Let *F* be a new built-in function object as defined in 11.3.4.
    b. The value of *F*'s length property is 2.
    c. Let *bc* be BoundFunctionCreate(*F*, «**this** value»).
    d. Set *collator*.[[boundCompare]] to *bc*.
3. Return *collator*.[[boundCompare]].

NOTE The function returned by [[Get]] is bound to this Collator object so that it can be passed directly to Array.prototype.sort or other functions.

The value of the [[Set]] attribute is **undefined**.

10.3.4 Collator Compare Functions

A Collator compare function is an anonymous function that takes two arguments, *x* and *y*, and performs the following steps:

1. Let *collator* be the **this** value.
2. Assert: Type(*collator*) is Object and *collator* has an [[initializedCollator]] internal slot whose value is `true`.
3. If *x* is not provided, let *x* be undefined.
4. If *y* is not provided, let *y* be undefined.
5. Let *X* be ToString(*x*).

6. ReturnIfAbrupt(*X*).
7. Let *Y* be ToString(*y*).
8. ReturnIfAbrupt(*Y*).
9. Return CompareStrings(*collator, X, Y*).

When the CompareStrings abstract operation is called with arguments *collator* (which must be an object initialized as a Collator), *x* and *y* (which must be String values), it returns a Number other than **NaN** that represents the result of a locale-sensitive String comparison of *x* with *y*. The two Strings are compared in an implementation-defined fashion. The result is intended to order String values in the sort order specified by the effective locale and collation options computed during construction of *collator*, and will be negative, zero, or positive, depending on whether *x* comes before *y* in the sort order, the Strings are equal under the sort order, or *x* comes after *y* in the sort order, respectively. String values must be interpreted as UTF-16 code unit sequences, and a surrogate pair (a code unit in the range 0xD800 to 0xDBFF followed by a code unit in the range 0xDC00 to 0xDFFF) within a string must be interpreted as the corresponding code point.

The sensitivity of *collator* is interpreted as follows:

- base: Only strings that differ in base letters compare as unequal. Examples: a ≠ b, a = á, a = A.
- accent: Only strings that differ in base letters or accents and other diacritic marks compare as unequal. Examples: a ≠ b, a ≠ á, a = A.
- case: Only strings that differ in base letters or case compare as unequal. Examples: a ≠ b, a = á, a ≠ A.
- variant: Strings that differ in base letters, accents and other diacritic marks, or case compare as unequal. Other differences may also be taken into consideration. Examples: a ≠ b, a ≠ á, a ≠ A.

NOTE In some languages, certain letters with diacritic marks are considered base letters. For example, in Swedish, “ö” is a base letter that’s different from “o”.

If the collator is set to ignore punctuation, then strings that differ only in punctuation compare as equal.

For the interpretation of options settable through extension keys, see Unicode Technical Standard 35.

The CompareStrings abstract operation with any given *collator* argument, if considered as a function of the remaining two arguments *x* and *y*, must be a consistent comparison function (as defined in ES2015, 22.1.3.24) on the set of all Strings.

The actual return values are implementation-defined to permit implementers to encode additional information in the value. The method is required to return 0 when comparing Strings that are considered canonically equivalent by the Unicode standard.

NOTE 1 It is recommended that the CompareStrings abstract operation be implemented following Unicode Technical Standard 10, Unicode Collation Algorithm (available at http://unicode.org/reports/tr10/), using tailorings for the effective locale and collation options of *collator*. It is recommended that implementations use the tailorings provided by the Common Locale Data Repository (available at http://cldr.unicode.org/).

NOTE 2 Applications should not assume that the behaviour of the CompareStrings abstract operation for Collator instances with the same resolved options will remain the same for different versions of the same implementation.

10.3.5 Intl.Collator.prototype.resolvedOptions ()

This function provides access to the locale and collation options computed during initialization of the object.

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal slot of this Collator object (see 10.4): locale, usage, sensitivity, ignorePunctuation, collation, as well as those properties shown in Table 1 whose keys are included in the `%Collator%`[[relevantExtensionKeys]] internal slot of the standard built-in object that is the initial value of Intl.Collator.

10.4 Properties of Intl.Collator Instances

Intl.Collator instances are ordinary objects that inherit properties from **`%CollatorPrototype%`**.

Intl.Collator instances and other objects that have been successfully initialized as a Collator have [[initializedIntlObject]] and [[initializedCollator]] internal slots whose values are **true**.

Objects that have been successfully initialized as a Collator also have several internal slots that are computed by the constructor:

- [[locale]] is a String value with the language tag of the locale whose localization is used for collation.
- [[usage]] is one of the String values **`"sort"`** or **`"search"`**, identifying the collator usage.
- [[sensitivity]] is one of the String values **`"base"`**, **`"accent"`**, **`"case"`**, or **`"variant"`**, identifying the collator's sensitivity.
- [[ignorePunctuation]] is a Boolean value, specifying whether punctuation should be ignored in comparisons.
- [[collation]] is a String value with the "type" given in Unicode Technical Standard 35 for the collation, except that the values **`"standard"`** and **`"search"`** are not allowed, while the value **`"default"`** is allowed.

Objects that have been successfully initialized as a Collator also have the following internal slots if the key corresponding to the name of the internal slot in Table 1 is included in the [[relevantExtensionKeys]] internal slot of Intl.Collator:

- [[numeric]] is a Boolean value, specifying whether numeric sorting is used.
- [[caseFirst]] is a String value; allowed values are specified in Table 1.

Finally, objects that have been successfully initialized as a Collator have a [[boundCompare]] internal slot that caches the function returned by the compare accessor (10.3.2).

11 NumberFormat Objects

11.1 The Intl.NumberFormat Constructor

The NumberFormat constructor is a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

11.1.1 InitializeNumberFormat (numberFormat, locales, options)

The abstract operation InitializeNumberFormat accepts the arguments *numberFormat* (which must be an object), *locales*, and *options*. It initializes *numberFormat* as a NumberFormat object.

The following steps are taken:

1. If *numberFormat* has an [[initializedIntlObject]] internal slot with value **true**, throw a **TypeError** exception.
2. Set *numberFormat*.[[initializedIntlObject]] to **true**.
3. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
4. ReturnIfAbrupt(*requestedLocales*).
5. If *options* is **undefined**, then
    a. Let *options* be ObjectCreate**`(%ObjectPrototype%)`**.
6. Else
    a. Let *options* be ToObject(*options*).
    b. ReturnIfAbrupt(*options*).
7. Let *opt* be a new Record.
8. Let *matcher* be GetOption(*options*, **`"localeMatcher"`**, **`"string"`**, «**`"lookup", "best fit"`**», **`"best fit"`**).
9. ReturnIfAbrupt(*matcher*).
10. Set *opt*.[[localeMatcher]] to *matcher*.

11. Let *localeData* be **%NumberFormat%.**[[localeData]].
12. Let *r* be ResolveLocale(**%NumberFormat%.**[[availableLocales]], *requestedLocales*, *opt*, **%NumberFormat%.**[[relevantExtensionKeys]], *localeData*).
13. Set *numberFormat*.[[locale]] to the value of *r*.[[locale]].
14. Set *numberFormat*.[[numberingSystem]] to the value of *r*.[[nu]].
15. Let *dataLocale* be *r*.[[dataLocale]].
16. Let *s* be GetOption(*options*, **"style"**, **"string"**, « **"decimal"**, **"percent"**, **"currency"**», **"decimal"**).
17. ReturnIfAbrupt(*s*).
18. Set *numberFormat*.[[style]] to *s*.
19. Let *c* be GetOption(*options*, **"currency"**, **"string"**, **undefined**, **undefined**).
20. ReturnIfAbrupt(*c*).
21. If *c* is not **undefined**, then
    a. If the result of IsWellFormedCurrencyCode(*c*), is **false**, then throw a **RangeError** exception.
22. If *s* is **"currency"** and *c* is **undefined**, throw a **TypeError** exception.
23. If *s* is **"currency"**, then
    a. Let *c* be converting *c* to upper case as specified in 6.1.
    b. Set *numberFormat*.[[currency]] to *c*.
    c. Let *cDigits* be CurrencyDigits(*c*)
24. Let *cd* be GetOption(*options*, **"currencyDisplay"**, **"string"**, «**"code"**, **"symbol"**, **"name"**», **"symbol"**).
25. ReturnIfAbrupt(*cd*).
26. If *s* is **"currency"**, set *numberFormat*.[[currencyDisplay]] to *cd*.
27. Let *mnid* be GetNumberOption(*options*, **"minimumIntegerDigits"**, 1, 21, 1).
28. ReturnIfAbrupt(*mnid*).
29. Set *numberFormat*.[[minimumIntegerDigits]] to *mnid*.
30. If *s* is **"currency"**, let *mnfdDefault* be *cDigits*; else let *mnfdDefault* be 0.
31. Let *mnfd* be GetNumberOption(*options*, **"minimumFractionDigits"**, 0, 20, *mnfdDefault*).
32. ReturnIfAbrupt(*mnfd*).
33. Set *numberFormat.*[[minimumFractionDigits]] to *mnfd*.
34. If *s* is **"currency"**, let *mxfdDefault* be max(*mnfd*, *cDigits*); else if *s* is **"percent"**, let *mxfdDefault* be max(*mnfd*, 0); else let *mxfdDefault* be max(*mnfd*, 3).
35. Let *mxfd* be GetNumberOption(*options*, **"maximumFractionDigits"**, *mnfd*, 20, *mxfdDefault*).
36. ReturnIfAbrupt(*mxfd*).
37. Set *numberFormat.*[[maximumFractionDigits]] to *mxfd*.
38. Let *mnsd* be Get(*options,* **"minimumSignificantDigits"**).
39. ReturnIfAbrupt(*mnsd*).
40. Let *mxsd* be Get(*options*, **"maximumSignificantDigits"**).
41. ReturnIfAbrupt(*mxsd*).
42. If *mnsd* is not **undefined** or *mxsd* is not **undefined**, then
    a. Let *mnsd* be GetNumberOption(*options*, **"minimumSignificantDigits"**, 1, 21, 1).
    b. ReturnIfAbrupt(*mnsd*).
    c. Let *mxsd* be GetNumberOption(*options*, **"maximumSignificantDigits"**, *mnsd*, 21, 21).
    d. ReturnIfAbrupt(*mxsd*).
    e. Set *numberFormat*.[[minimumSignificantDigits]] to *mnsd.*
    f. Set *numberFormat*.[[maximumSignificantDigits]] to *mxsd*.
43. Let *g* be GetOption(*options*, **"useGrouping"**, **"boolean"**, **undefined**, **true**).
44. ReturnIfAbrupt(*g*).
45. Set *numberFormat*.[[useGrouping]] to *g*.
46. Let *dataLocaleData* be Get(*localeData, dataLocale).*
47. Let *patterns* be Get(*dataLocaleData*, **"patterns")**.
48. Assert: *patterns* is an object (see 11.2.3).
49. Let *stylePatterns* be Get(*patterns*, *s*).
50. Set *numberFormat.*[[positivePattern]] to Get(*stylePatterns*, **"positivePattern")**.
51. Set *numberFormat.*[[negativePattern]] to Get(*stylePatterns*, **"negativePattern")**.
52. Set *numberFormat.*[[boundFormat]] to **undefined**.
53. Set *numberFormat.*[[initializedNumberFormat]] to **true**.
54. Return *numberFormat*.

When the abstract operation CurrencyDigits is called with an argument *currency* (which must be an upper case String value), the following steps are taken:

1. If the ISO 4217 currency and funds code list contains *currency* as an alphabetic code, then return the minor unit value corresponding to the *currency* from the list; else return 2.

11.1.2 Intl.NumberFormat([locales [, options]])

When the **`Intl.NumberFormat`** function is called with optional arguments the following steps are taken:

1. If NewTarget is **undefined,** let *newTarget* be the active function object, else let *newTarget* be NewTarget
2. Let *numberFormat* be OrdinaryCreateFromConstructor(*newTarget*, **`%NumberFormatPrototype%`**).
3. ReturnIfAbrupt(*numberFormat*).
4. Return InitializeNumberFormat(*numberFormat, locales, options*).

11.2 Properties of the Intl.NumberFormat Constructor

Besides the internal slots and the **`length`** property (whose value is 0), the Intl.NumberFormat constructor has the following properties:

11.2.1 Intl.NumberFormat.prototype

The value of **`Intl.NumberFormat.prototype`** is **`%NumberFormatPrototype%`**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

11.2.2 Intl.NumberFormat.supportedLocalesOf (locales [, options])

When the **`supportedLocalesOf`** method of **`%NumberFormat%`** is called, the following steps are taken:

1. Let *availableLocales* be **`%NumberFormat%`**.[[*availableLocales*]].
2. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
3. Return SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the **`length`** property of the **`supportedLocalesOf`** method is **`1`**.

11.2.3 Internal slots

The value of the [[availableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[relevantExtensionKeys]] internal slot is **`["nu"]`**.

NOTE Unicode Technical Standard 35 describes two locale extension keys that are relevant to number formatting, "nu" for numbering system and "cu" for currency. Intl.NumberFormat, however, requires that the currency of a currency format is specified through the currency property in the options objects.

The value of the [[localeData]] internal slot is implementation defined within the constraints described in 9.1 and the following additional constraints:

- The array that is the value of the "nu" property of any locale property of [[localeData]] must not include the values "native", "traditio", or "finance".
- [[localeData]][locale] must have a patterns property for all locale values. The value of this property must be an object, which must have properties with the names of the three number format styles: **`"decimal"`**, **`"percent"`**, and **`"currency"`**. Each of these properties in turn must be an object with the properties positivePattern and negativePattern. The value of these properties must be string values that contain a substring **`"{number}"`**; the values within the currency property must also contain a substring **`"{currency}"`**. The pattern strings must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.

NOTE It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at http://cldr.unicode.org/).

11.3 Properties of the Intl.NumberFormat Prototype Object

The Intl.NumberFormat prototype object is itself an Intl.NumberFormat instance as specified in 11.4, whose internal slots are set as if it had been constructed by the expression Construct(`%NumberFormat%`).

In the following descriptions of functions that are properties or [[Get]] attributes of properties of `%NumberFormatPrototype%`, the phrase "this NumberFormat object" refers to the object that is the this value for the invocation of the function; a **TypeError** exception is thrown if the this value is not an object or an object that does not have an [[initializedNumberFormat]] internal slot with value **true**.

11.3.1 Intl.NumberFormat.prototype.constructor

The initial value of `Intl.NumberFormat.prototype.constructor` is `%NumberFormat%`.

11.3.2 Intl.NumberFormat.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

11.3.3 Intl.NumberFormat.prototype.format

This named accessor property returns a function that formats a number according to the effective locale and the formatting options of this NumberFormat object.

The value of the [[Get]] attribute is a function that takes the following steps:

1. Let *nf* be this NumberFormat object.
2. If *nf*.[[boundFormat]] is **undefined**, then
    a. Let *F* be a new built-in function object as defined in 11.3.4.
    b. The value of *F*'s `length` property is 1.
    c. Let *bf* be BoundFunctionCreate(*F*, «**this** value»).
    d. Set *nf*.[[boundFormat]] to *bf*.
3. Return *nf*.[[boundFormat]].

NOTE The function returned by [[Get]] is bound to this NumberFormat object so that it can be passed directly to Array.prototype.map or other functions.

The value of the [[Set]] attribute is **undefined**.

11.3.4 Number Format Functions

A NumberFormat format function is an anonymous function that takes one argument *value*, and performs the following steps:

1. Let *nf* be the **this** value.
2. Assert: Type(*nf*) is Object and *nf* has an [[initializedNumberFormat]] internal slot whose value is **`true`**.
3. If *value* is not provided, let *value* be **undefined**.
4. Let *x* be ToNumber(*value*).
5. ReturnIfAbrupt(*x*).
6. Return FormatNumber(*nf*, *x*).

When the FormatNumber abstract operation is called with arguments *numberFormat* (which must be an object initialized as a NumberFormat) and *x* (which must be a Number value), it returns a String value representing *x* according to the effective locale and the formatting options of *numberFormat*.

The computations rely on String values and locations within numeric strings that are dependent upon the implementation and the effective locale of *numberFormat* ("ILD") or upon the implementation, the effective locale, and the numbering system of *numberFormat* ("ILND"). The ILD and ILND Strings mentioned, other than those for currency names, must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.

NOTE It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at http://cldr.unicode.org/).

The following steps are taken:

1. Let *negative* be **false**.
2. If the result of isFinite(*x*) is **false**, then
    a. If *x* is **NaN**,
        i. Let *n* be an ILD String value indicating the NaN value.
    b. Else
        i. Let *n* be an ILD String value indicating infinity.
        ii. If *x* < 0, then let *negative* be **true**.
3. Else
    a. If *x* < 0, then
        i. Let *negative* be **true**.
        ii. Let *x* be -*x*.
    b. If the value of *numberFormat*.[[style]] is **"percent"**, let *x* be 100 × *x*.
    c. If the *numberFormat*.[[minimumSignificantDigits]] and *numberFormat*.[[maximumSignificantDigits]] are present, then
        i. Let *n* be ToRawPrecision(*x*, *numberFormat*.[[minimumSignificantDigits]], *numberFormat*.[[maximumSignificantDigits]]).
    d. Else
        i. Let *n* be ToRawFixed(*x, numberFormat*.[[minimumIntegerDigits]], *numberFormat*.[[minimumFractionDigits]], *numberFormat*.[[maximumFractionDigits]]).
    e. If the value of the *numberFormat*.[[numberingSystem]] matches one of the values in the "Numbering System" column of Table 2 below, then
        i. Let *digits* be an array whose 10 String valued elements are the UTF-16 string representations of the 10 digits specified in the "Digits" column of the matching row in Table 2.
        ii. Replace each *digit* in *n* with the value of *digits*[*digit*].
    f. Else use an implementation dependent algorithm to map *n* to the appropriate representation of *n* in the given numbering system.
    g. If *n* contains the character **"."**, then replace it with an ILND String representing the decimal separator.
    h. If the value of the *numberFormat*.[[useGrouping]] is **true**, then insert an ILND String representing a grouping separator into an ILND set of locations within the integer part of *n*.
4. If *negative* is **true**, then
    a. Let *result* be the value of *numberFormat*.[[negativePattern]].
5. Else,
    a. Let *result* be the value of *numberFormat*.[[positivePattern]].
6. Replace the substring **"{number}"** within *result* with *n*.
7. If the value of the *numberFormat*.[[style]] is **"currency"**, then
    a. Let *currency* be the value of *numberFormat*.[[currency]].
    b. If *numberFormat*.[[currencyDisplay]] is **"code"**, then
        i. Let *cd* be *currency*.
    c. Else, if *numberFormat*.[[currencyDisplay]] is **"symbol"**, then
        i. Let *cd* be an ILD string representing *currency* in short form. If the implementation does not have such a representation of *currency*, then use *currency* itself.
    d. Else, if *numberFormat*.[[currencyDisplay]] is **"name"**, then
        i. Let *cd* be an ILD string representing *currency* in long form. If the implementation does not have such a representation of *currency*, then use *currency* itself.
    e. Replace the substring **"{currency}"** within *result* with *cd*.
8. Return *result*.

When the ToRawPrecision abstract operation is called with arguments *x* (which must be a finite non-negative number), *minPrecision*, and *maxPrecision* (both must be integers between 1 and 21) the following steps are taken:

1. Let *p* be *maxPrecision*.
2. If $x = 0$, then
   a. Let *m* be the String consisting of *p* occurrences of the character `"0"`.
   b. Let *e* be 0.
3. Else
   a. Let *e* and *n* be integers such that $10^{p-1} \leq n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-p+1}$ is larger.
   b. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If $e \geq p$, then
   a. Return the concatenation of *m* and e-p+1 occurrences of the character `"0"`.
5. If $e = p$-1, then
   a. Return *m*.
6. If $e \geq 0$, then
   a. Let *m* be the concatenation of the first e+1 characters of *m*, the character `"."`, and the remaining p–(e+1) characters of *m*.
7. If $e < 0$, then
   a. Let *m* be the concatenation of the String `"0."`, –(e+1) occurrences of the character `"0"`, and the string *m*.
8. If *m* contains the character `"."`, and *maxPrecision* > *minPrecision*, then
   a. Let *cut* be *maxPrecision* – *minPrecision*.
   b. Repeat while *cut* > 0 and the last character of *m* is `"0"`:
      i. Remove the last character from *m*.
      ii. Decrease *cut* by 1.
   c. If the last character of *m* is `"."`, then
      i. Remove the last character from *m*.
9. Return *m*.

When the ToRawFixed abstract operation is called with arguments *x* (which must be a finite non-negative number), *minInteger* (which must be an integer between 1 and 21), *minFraction*, and *maxFraction* (which must be integers between 0 and 20) the following steps are taken:

1. Let *f* be *maxFraction*.
2. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such *n*, pick the larger *n*.
3. If $n = 0$, let *m* be the String `"0"`. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If $f \neq 0$, then
   a. Let *k* be the number of characters in *m*.
   b. If $k \leq f$, then
      i. Let *z* be the String consisting of f+1–k occurrences of the character `"0"`.
      ii. Let *m* be the concatenation of Strings *z* and *m*.
      iii. Let k=f+1.
   c. Let *a* be the first k–f characters of m, and let *b* be the remaining *f* characters of *m*.
   d. Let *m* be the concatenation of the three Strings *a*, ".", and *b*.
   e. Let *int* be the number of characters in *a*.
5. Else let *int* be the number of characters in *m*.
6. Let *cut* be *maxFraction* – *minFraction*.
7. Repeat while *cut* > 0 and the last character of *m* is `"0"`:
   a. Remove the last character from *m*.
   b. Decrease *cut* by 1.
8. If the last character of *m* is `"."`, then
   a. Remove the last character from *m*.
9. If *int* < *minInteger*, then
   a. Let *z* be the String consisting of *minInteger*–*int* occurrences of the character `"0"`.

b. Let *m* be the concatenation of Strings *z* and *m*.

10. Return *m*.

Table 2 – Numbering systems with simple digit mappings

Numbering System	Digits
arab	U+0660 to U+0669
arabext	U+06F0 to U+06F9
bali	U+1B50 to U+1B59
beng	U+09E6 to U+09EF
deva	U+0966 to U+096F
fullwide	U+FF10 to U+FF19
gujr	U+0AE6 to U+0AEF
guru	U+0A66 to U+0A6F
hanidec	U+3007, U+4E00, U+4E8C, U+4E09, U+56DB, U+4E94, U+516D, U+4E03, U+516B, U+4E5D
khmr	U+17E0 to U+17E9
knda	U+0CE6 to U+0CEF
laoo	U+0ED0 to U+0ED9
latn	U+0030 to U+0039
limb	U+1946 to U+194F
mlym	U+0D66 to U+0D6F
mong	U+1810 to U+1819
mymr	U+1040 to U+1049
orya	U+0B66 to U+0B6F
tamldec	U+0BE6 to U+0BEF
telu	U+0C66 to U+0C6F
thai	U+0E50 to U+0E59
tibt	U+0F20 to U+0F29

11.3.5 Intl.NumberFormat.prototype.resolvedOptions ()

This function provides access to the locale and formatting options computed during initialization of the object.

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal slot of this NumberFormat object (see 11.4): locale, numberingSystem, style, currency, currencyDisplay, minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, maximumSignificantDigits, and useGrouping. Properties whose corresponding internal slots are not present are not assigned.

11.4 Properties of Intl.NumberFormat Instances

Intl.NumberFormat instances inherit properties from `%NumberFormatPrototype%`.

Intl.NumberFormat instances and other objects that have been successfully initialized as a NumberFormat have [[initializedIntlObject]] and [[initializedNumberFormat]] internal slots whose values are **true**.

Objects that have been successfully initialized as a NumberFormat object also have several internal slots that are computed by the constructor:

- [[locale]] is a String value with the language tag of the locale whose localization is used for formatting.
- [[numberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
- [[style]] is one of the String values `"decimal"`, `"currency"`, or `"percent"`, identifying the number format style used.
- [[currency]] is a String value with the currency code identifying the currency to be used if formatting with the `"currency"` style. It is only present when [[style]] has the value `"currency"`.
- [[currencyDisplay]] is one of the String values `"code"`, `"symbol"`, or `"name"`, specifying whether to display the currency as an ISO 4217 alphabetic currency code, a localized currency symbol, or a localized currency name if formatting with the `"currency"` style. It is only present when [[style]] has the value `"currency"`.
- [[minimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used. Numbers will be padded with leading zeroes if necessary.
- [[minimumFractionDigits]] and [[maximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.
- [[minimumSignificantDigits]] and [[maximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be shown. Either none or both of these properties are present; if they are, they override minimum and maximum integer and fraction digits – the formatter uses however many integer and fraction digits are required to display the specified number of significant digits.
- [[useGrouping]] is a Boolean value indicating whether a grouping separator should be used.
- [[positivePattern]] and [[negativePattern]] are String values as described in 11.3.2.

Finally, objects that have been successfully initialized as a NumberFormat have a [[boundFormat]] internal slot that caches the function returned by the format accessor (11.3.2).

12 DateTimeFormat Objects

12.1 The Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor is a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

12.1.1 InitializeDateTimeFormat (dateTimeFormat, locales, options)

The abstract operation InitializeDateTimeFormat accepts the arguments *dateTimeFormat* (which must be an object), *locales*, and *options*. It initializes *dateTimeFormat* as a DateTimeFormat object.

Several DateTimeFormat algorithms use values from the following table, which provides property names and allowable values for the components of date and time formats:

Table 3 – Components of date and time formats

Property	Values
weekday	"narrow", "short", "long"
era	"narrow", "short", "long"
year	"2-digit", "numeric"
month	"2-digit", "numeric", "narrow", "short", "long"
day	"2-digit", "numeric"
hour	"2-digit", "numeric"
minute	"2-digit", "numeric"
second	"2-digit", "numeric"
timeZoneName	"short", "long"

The following steps are taken:

1. If *dateTimeFormat*.[[initializedIntlObject]] is **true**, throw a **TypeError** exception.
2. Set *dateTimeFormat*.[[initializedIntlObject]] to **true**.
3. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
4. ReturnIfAbrupt(*requestedLocales*),
5. Let *options* be ToDateTimeOptions(*options*, **`"any"`**, **`"date"`**).
6. ReturnIfAbrupt(*options*).
7. Let *opt* be a new Record.
8. Let *matcher* be GetOption(*options*, **`"localeMatcher"`**, **`"string"`**, «**`"lookup"`**, **`"best fit"`**», **`"best fit"`**).
9. ReturnIfAbrupt(*matcher*).
10. Set *opt*.[[localeMatcher]] to *matcher*.
11. Let *localeData* be the value of **`%DateTimeFormat%`**.[[localeData]].
12. Let *r* be ResolveLocale(**`%DateTimeFormat%`**.[[availableLocales]], *requestedLocales*, *opt*, **`%DateTimeFormat%`**.[[relevantExtensionKeys]], *localeData*).
13. Set *dateTimeFormat*.[[locale]] to the value of *r*.[[locale]].
14. Set *dateTimeFormat*.[[calendar]] to the value of *r*.[[ca]].
15. Set *dateTimeFormat*.[[numberingSystem]] to the value of *r*.[[nu]].
16. Let *dataLocale* be the value of *r*.[[dataLocale]].
17. Let *tz* be Get(*options,* **`"timeZone")`**.
18. ReturnIfAbrupt(*tz*).
19. If *tz* is not **undefined**, then
    a. Let *tz* be ToString(*tz*).
    b. ReturnIfAbrupt(*tz*).
    c. If the result of IsValidTimeZoneName(*tz*) is **false**, then
        i. Throw a **RangeError** exception.
    d. Let *tz* be CanonicalizeTimeZoneName(*tz*).
20. Else,
    a. Let *tz* be DefaultTimeZone().
21. Set *dateTimeFormat*.[[timeZone]] to *tz*.
22. Let *opt* be a new Record.
23. For each row of Table 3, except the header row, do:
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *value* be GetOption(*options*, *prop*, **`"string"`**, «the strings given in the Values column of the row», **undefined**).
    c. ReturnIfAbrupt(*value*).
    d. Set *opt*.[[<*prop*>]] to *value*.
24. Let *dataLocaleData* be Get(*localeData, dataLocale*).
25. Let *formats* be Get(*dataLocaleData,* **`"formats"`**).
26. Let *matcher* be GetOption(*options*, **`"formatMatcher"`**, **`"string"`**, «**`"basic"`**, **`"best fit"`**», **`"best fit"`**).
27. ReturnIfAbrupt(*matcher*).
28. If *matcher* is **`"basic"`**, then
    a. Let *FormatMatcher* be the abstract operation BasicFormatMatcher.
29. Else
    a. Let *FormatMatcher* be the abstract operation BestFitFormatMatcher.
30. Let *bestFormat* be *FormatMatcher*(opt, formats).
31. For each row in Table 3, except the header row, do
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *p* be Get(*bestFormat, prop*).
    c. If *p* not **undefined**, then
        i. Set *dateTimeFormat*.[[<*prop*>]] to *p*.
32. Let *hr12* be GetOption(*options*, **`"hour12"`**, **`"boolean"`**, **undefined**, **undefined**).
33. ReturnIfAbrupt(*hr12*).
34. If *dateTimeFormat* has an internal slot [[hour]], then
    a. If *hr12* is **undefined**, then
        i. Let *hr12* be Get(*dataLocaleData*, **`"hour12"`**).
    b. Set *dateTimeFormat*.[[hour12]] to *hr12*.

c. If *hr12* is **true**, then
   i. Let *hourNo0* be Get(*dataLocaleData*, **"hourNo0"**).
   ii. Set *dateTimeFormat*.[[hourNo0]] to *hourNo0*.
   iii. Let *pattern* be Get(*bestFormat*, **"pattern12"**).
d. Else
   i. Let *pattern* be Get(*bestFormat*, **"pattern"**).

35. Else
    a. Let *pattern* be Get(*bestFormat*, **"pattern"**).
36. Set *dateTimeFormat*.[[pattern]] to *pattern*.
37. Set *dateTimeFormat*.[[boundFormat]] to **undefined**.
38. Set *dateTimeFormat*.[[initializedDateTimeFormat]] to **true**.
39. Return *dateTimeFormat*.

When the ToDateTimeOptions abstract operation is called with arguments *options*, *required*, and *defaults*, the following steps are taken:

1. If *options* is **undefined**, then let *options* be **null**, else let *options* be ToObject(*options*).
2. ReturnIfAbrupt(*options*).
3. Let *options* be ObjectCreate(*options*).
4. Let *needDefaults* be **true**.
5. If *required* is **"date"** or **"any"**,
   a. For each of the property names **"weekday"**, **"year"**, **"month"**, **"day"**:
      i. Let *prop* be the property name.
      ii. Let *value* be Get(*options*, *prop*).
      iii. ReturnIfAbrupt(*value*).
      iv. If *value* is not **undefined**, then let *needDefaults* be **false**.
6. If *required* is **"time"** or **"any"**,
   a. For each of the property names **"hour"**, **"minute"**, **"second"**:
      i. Let *prop* be the property name.
      ii. Let *value* be Get(*options*, *prop*).
      iii. ReturnIfAbrupt(*value*).
      iv. If *value* is not **undefined**, then let *needDefaults* be **false**.
7. If *needDefaults* is **true** and *defaults* is either **"date"** or **"all"**, then
   a. For each of the property names **"year"**, **"month"**, **"day"**:
      i. Let *status* be CreateDatePropertyOrThrow(*options*, *prop*, **"numeric"**).
      ii. ReturnIfAbrupt(*status*).
8. If *needDefaults* is **true** and *defaults* is either **"time"** or **"all"**, then
   a. For each of the property names **"hour"**, **"minute"**, **"second"**:
      i. Let *status* be CreateDatePropertyOrThrow(*options*, *prop*, **"numeric"**).
      ii. ReturnIfAbrupt(*status*).
9. Return *options*.

When the BasicFormatMatcher abstract operation is called with two arguments *options* and *formats*, the following steps are taken:

1. Let *removalPenalty* be 120.
2. Let *additionPenalty* be 20.
3. Let *longLessPenalty* be 8.
4. Let *longMorePenalty* be 6.
5. Let *shortLessPenalty* be 6.
6. Let *shortMorePenalty* be 3.
7. Let *bestScore* be -**Infinity**.
8. Let *bestFormat* be **undefined**.
9. Let *k* be 0.
10. Assert: *formats* is an Array object.
11. Let *len* be Get(*formats*, **"length"**).

12. Repeat while *k* < *len*:
    a. Let *format* be Get(*formats,* ToString(*k*)).
    b. Let *score* be 0.
    c. For each *property* shown in Table 3:
        i. Let *optionsProp* be *options*.[[<*property*>]].
        ii. Let *formatProp* be Get(*format, property*).
        iii. If *optionsProp* is **undefined** and *formatProp* is not **undefined**, then decrease *score* by *additionPenalty*.
        iv. Else if *optionsProp* is not **undefined** and *formatProp* is **undefined**, then decrease *score* by *removalPenalty*.
        v. Else if *optionsProp* ≠ *formatProp*,
            1. Let *values* be the array **`["2-digit", "numeric", "narrow", "short", "long"]`**.
            2. Let *optionsPropIndex* be the index of *optionsProp* within *values*.
            3. Let *formatPropIndex* be the index of *formatProp* within *values*.
            4. Let *delta* be max(min(*formatPropIndex* - *optionsPropIndex*, 2), -2).
            5. If *delta* = 2, decrease *score* by *longMorePenalty*.
            6. Else if *delta* = 1, decrease *score* by *shortMorePenalty*.
            7. Else if *delta* = -1, decrease *score* by *shortLessPenalty*.
            8. Else if *delta* = -2, decrease *score* by *longLessPenalty*.
    d. If *score* > *bestScore*,
        i. Let *bestScore* be *score*.
        ii. Let *bestFormat* be *format*.
    e. Increase *k* by 1.
13. Return *bestFormat*.

When the BestFitFormatMatcher abstract operation is called with two arguments *options* and *formats*, it performs implementation dependent steps, which should return a set of component representations that a typical user of the selected locale would perceive as at least as good as the one returned by BasicFormatMatcher.

12.1.2 Intl.DateTimeFormat([locales [, options]])

When the **`Intl.DateTimeFormat`** function is called with optional arguments *locales* and *options* the following steps are taken:

1. If NewTarget is **undefined**, let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *dateTimeFormat* be OrdinaryCreateFromConstructor(*newTarget*, **`%DateTimeFormatPrototype%`**).
3. ReturnIfAbrupt(*dateTimeFormat*).
4. Return InitializeDateTimeFormat(*dateTimeFormat, locales, options*).

12.2 Properties of the Intl.DateTimeFormat Constructor

Besides the internal slots and the **`length`** property (whose value is 0), the Intl.DateTimeFormat constructor has the following properties:

12.2.1 Intl.DateTimeFormat.prototype

The value of **`Intl.DateTimeFormat.prototype`** is **`%DateTimeFormatPrototype%`**.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

12.2.2 Intl.DateTimeFormat.supportedLocalesOf (locales [, options])

When the `supportedLocalesOf` method of Intl.DateTimeFormat is called, the following steps are taken:

1. Let *availableLocales* be the value of `%DateTimeFormat%`.[[availableLocales]].
2. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
3. ReturnIfAbrupt(*requestedLocales*).
4. Return *supportedLocales* be SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the `length` property of the `supportedLocalesOf` method is `1`.

12.2.3 Internal slots

The value of the [[availableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[relevantExtensionKeys]] internal slot is `["ca", "nu"]`.

NOTE Unicode Technical Standard 35 describes three locale extension keys that are relevant to date and time formatting, "ca" for calendar, "tz" for time zone, and implicitly "nu" for the numbering system of the number format used for numbers within the date format. DateTimeFormat, however, requires that the time zone is specified through the timeZone property in the options objects.

The value of the [[localeData]] internal slot is implementation defined within the constraints described in 9.1 and the following additional constraints:

- The array that is the value of the "nu" property of any locale property of [[localeData]] must not include the values "native", "traditio", or "finance".
- [[localeData]][[locale] must have hour12 and hourNo0 properties with Boolean values for all locale values.
- [[localeData]][[locale] must have a formats property for all locale values. The value of this property must be an array of objects, each of which has a subset of the properties shown in Table 3, where each property must have one of the values specified for the property in Table 3. Multiple objects in an array may use the same subset of the properties as long as they have different values for the properties. The following subsets must be available for each locale:
  - weekday, year, month, day, hour, minute, second
  - weekday, year, month, day
  - year, month, day
  - year, month
  - month, day
  - hour, minute, second
  - hour, minute

  Each of the objects must also have a pattern property, whose value is a String value that contains for each of the date and time format component properties of the object a substring starting with "{", followed by the name of the property, followed by "}". If the object has an hour property, it must also have a pattern12 property, whose value is a String value that, in addition to the substrings of the pattern property, contains a substring "{ampm}".

EXAMPLE An implementation might include the following object as part of its English locale data: {hour: "numeric", minute: "2-digit", second: "2-digit", pattern: "{hour}:{minute}:{second}", pattern12: "{hour}:{minute}:{second} {ampm}"}.

NOTE It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at http://cldr.unicode.org/).

12.3 Properties of the Intl.DateTimeFormat Prototype Object

The Intl.DateTimeFormat prototype object is itself an `%DateTimeFormat%` instance, whose internal slots are set as if it had been constructed by the expression `Construct(%DateTimeFormat%)`.

In the following descriptions of functions that are properties or [[Get]] attributes of properties of the Intl.DateTimeFormat prototype object, the phrase "this DateTimeFormat object" refers to the object that is the this value for the invocation of the function; a **TypeError** exception is thrown if the this value is not an object or an object that does not have an [[initializedDateTimeFormat]] internal slot with value **true**.

12.3.1 Intl.DateTimeFormat.prototype.constructor

The initial value of `Intl.DateTimeFormat.prototype.constructor` is `%DateTimeFormat%`.

12.3.2 Intl.DateTimeFormat.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

12.3.3 Intl.DateTimeFormat.prototype.format

This named accessor property returns a function that formats a date according to the effective locale and the formatting options of this DateTimeFormat object.

The value of the [[Get]] attribute is a function that takes the following steps:

1. Let *dtf* be this DateTimeFormat object.
2. ReturnIfAbrupt(*dtf*).
3. If the [[boundFormat]] internal slot of this DateTimeFormat object is **undefined**, then
   a. Let *F* be a new built-in function object as defined in 12.3.4.
   b. The value of *F*'s `length` property is 1.
   c. Let *bf* be BoundFunctionCreate(*F, «***this** value»).
   d. Set *dtf*.[[boundFormat]] to *bf*.
4. Return *dtf*.[[boundFormat]].

12.3.4 DateTime Format Functions

A DateTime format function is an anonymous function that optionally takes an argument *date*. It performs the following steps:

1. Let *dtf* be the **this** value.
7. Assert: Type(*dtf*) is Object and *dtf* has an [[initializedDateTimeFormat]] internal slot whose value is `true`.
8. If *date* is not provided or is **undefined**, then
   a. Let *x* be `%Date_now%`().
9. Else
   b. Let *x* be ToNumber(*date*).
   c. ReturnIfAbrupt(*x*).
10. Return FormatDateTime(*dtf, x*).

NOTE The function returned by [[Get]] is bound to this DateTimeFormat object so that it can be passed directly to Array.prototype.map or other functions.

The value of the [[Set]] attribute is **undefined**.

When the FormatDateTime abstract operation is called with arguments *dateTimeFormat* (which must be an object initialized as a DateTimeFormat) and *x* (which must be a Number value), it returns a String value representing *x* (interpreted as a time value as specified in ES2015, 20.3.1.1) according to the effective locale and the formatting options of *dateTimeFormat*.

1. If *x* is not a finite Number, then throw a **RangeError** exception.
11. Let *locale* be the value of *dateTimeFormat*.[[locale]].
12. Let *nf* be Construct(`%NumberFormat%, «[locale], {useGrouping: false}»)`.
13. ReturnIfAbrupt(*nf*).
14. Let *nf2* be Construct(`%NumberFormat%, «[locale], {minimumIntegerDigits: 2, useGrouping: false}»)`.
15. ReturnIfAbrupt(*nf2*).
16. Let *tm* be ToLocalTime(*x*, *dateTimeFormat*.[[calendar]], *dateTimeFormat*.[[timeZone]]).
17. Let *result* be the value of the *dateTimeFormat*.[[pattern]].
18. For each row of Table 3, except the header row, do:
    a. If *dateTimeFormat* has an internal slot with the name given in the Property column of the row, then
        i. Let *p* be the name given in the Property column of the row.
        ii. Let *f* be the value of the [[<*p*>]] internal slot of *dateTimeFormat*.
        iii. Let *v* be the value of *tm*.[[<*p*>]].
        iv. If *p* is **`"year"`** and $v \leq 0$, let *v* be 1 - *v*.
        v. If *p* is **`"month"`**, increase *v* by 1.
        vi. If *p* is **`"hour"`** and the value of *dateTimeFormat*.[[hour12]] is **true**, then
            1. Let *v* be *v* modulo 12.
            2. If *v* is equal to the value of *tm*.[[<*p*>]], let *pm* be **false**; else let *pm* be **true**.
            3. If *v* is 0 and the value of *dateTimeFormat*.[[hourNo0]] is **true**, let *v* be 12.
        vii. If *f* is **`"numeric"`**, then
            1. Let *fv* be FormatNumber(*nf*, *v*).
        viii. Else if *f* is **`"2-digit"`**, then
            1. Let *fv* be FormatNumber(*nf2*, *v*).
            2. If the length of *fv* is greater than 2, let *fv* be the substring of *fv* containing the last two characters.
        ix. Else if *f* is **`"narrow"`**, **`"short"`**, or **`"long"`**, then let *fv* be a String value representing *f* in the desired form; the String value depends upon the implementation and the effective locale and calendar of *dateTimeFormat*. If *p* is **`"month"`**, then the String value may also depend on whether *dateTimeFormat* has a [[day]] internal slot. If *p* is **`"timeZoneName"`**, then the String value may also depend on the value of the [[inDST]] field of *tm*, and if the implementation does not have a localized representation of *f*, then use *f* itself.
        x. Replace the substring of *result* that consists of `"{"`, *p*, and `"}"`, with *fv*.
19. If *dateTimeFormat*.[[hour12]] is **true**, then
    b. If *pm* is **true**, then
        i. Let *fv* be an implementation and locale dependent String value representing “post meridiem”;
    c. Else
        i. Let *fv* be an implementation and locale dependent String value representing “ante meridiem”.
    d. Replace the substring of *result* that consists of **`"{ampm}"`**, with *fv*.
20. Return *result*.

NOTE It is recommended that implementations use the locale and calendar dependent strings provided by the Common Locale Data Repository (available at http://cldr.unicode.org/), and use CLDR “abbreviated” strings for DateTimeFormat “short” strings, and CLDR “wide” strings for DateTimeFormat “long” strings.

When the ToLocalTime abstract operation is called with arguments *date*, *calendar*, and *timeZone*, the following steps are taken:

1. Apply calendrical calculations on *date* for the given *calendar* and *timeZone* to produce weekday, era, year, month, day, hour, minute, second, and inDST values. The calculations should use best available information about the specified *calendar* and *timeZone*, including current and historical information about time zone offsets from UTC and daylight saving time rules. If the *calendar* is "gregory", then the calculations must match the algorithms specified in ES2015, 20.3.1.
2. Return a Record with fields [[weekday]], [[era]], [[year]], [[month]], [[day]], [[hour]], [[minute]], [[second]], and [[inDST]], each with the corresponding calculated value.

NOTE It is recommended that implementations use the time zone information of the IANA Time Zone Database.

12.3.5 Intl.DateTimeFormat.prototype.resolvedOptions ()

This function provides access to the locale and formatting options computed during initialization of the object.

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal slot of this DateTimeFormat object (see 12.4): locale, calendar, numberingSystem, timeZone, hour12, weekday, era, year, month, day, hour, minute, second, and timeZoneName. Properties whose corresponding internal slots are not present are not assigned.

NOTE In this version of the ECMAScript 2015 Internationalization API, the timeZone property will be the name of the default time zone if no timeZone property was provided in the options object provided to the Intl.DateTimeFormat constructor. The previous version left the timeZone property undefined in this case.

12.4 Properties of Intl.DateTimeFormat Instances

Intl.DateTimeFormat instances inherit properties from `%DateTimeFormatPrototype%`.

Intl.DateTimeFormat instances and other objects that have been successfully initialized as a DateTimeFormat object have [[initializedIntlObject]] and [[initializedDateTimeFormat]] internal slots whose values are **true**.

Objects that have been successfully initialized as a DateTimeFormat also have several internal slots that are computed by the constructor:

- [[locale]] is a String value with the language tag of the locale whose localization is used for formatting.
- [[calendar]] is a String value with the "type" given in Unicode Technical Standard 35 for the calendar used for formatting.
- [[numberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
- [[timeZone]] is a String value with the IANA time zone name of the time zone used for formatting.
- [[weekday]], [[era]], [[year]], [[month]], [[day]], [[hour]], [[minute]], [[second]], [[timeZoneName]] are each either absent, indicating that the component is not used for formatting, or one of the String values given in Table 3, indicating how the component should be presented in the formatted output.
- [[hour12]] is a Boolean value indicating whether 12-hour format (**true**) or 24-hour format (**false**) should be used. It is only present when [[hour]] is also present.
- [[hourNo0]] is a Boolean value indicating whether hours from 1 to 12 (**true**) or from 0 to 11 (**false**) should be used. It is only present when [[hour12]] is also present and has the value **true**.
- [[pattern]] is a String value as described in 12.3.2.

Finally, objects that have been successfully initialized as a DateTimeFormat have a [[boundFormat]] internal slot that caches the function returned by the format accessor (12.3.2).

13 Locale Sensitive Functions of the ECMAScript Language Specification

The ECMAScript Language Specification, edition 6 or successor, describes several locale sensitive functions. An ECMAScript implementation that implements this Internationalization API Specification shall implement these functions as described here.

NOTE The Collator, NumberFormat, or DateTimeFormat objects created in the algorithms in this clause are only used within these algorithms. They are never directly accessed by ECMAScript code and need not actually exist within an implementation.

13.1 Properties of the String Prototype Object

13.1.1 String.prototype.localeCompare (that [, locales [, options]])

This definition supersedes the definition provided in ES2015, 21.1.3.10.

When the `localeCompare` method is called with argument *that* and optional arguments *locales*, and *options*, the following steps are taken:

1. Let *O* be RequireObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *That* be ToString(*that*).
5. ReturnIfAbrupt(*That*).
6. Let *collator* be Construct(`%Collator%, «locales, options»)`.
7. ReturnIfAbrupt(*collator*).
8. Return CompareStrings(*collator, S, That*).

The value of the `length` property of the `localeCompare` method is `1`.

NOTE 1 The `localeCompare` method itself is not directly suitable as an argument to `Array.prototype.sort` because the latter requires a function of two arguments.

NOTE 2 The `localeCompare` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

13.1.2 String.prototype.toLocaleLowerCase ([locales])

This definition supersedes the definition provided in ES2015, 21.1.3.20.

This function interprets a string value as a sequence of code points, as described in ES2015, 6.1.4. The following steps are taken:

1. Let *O* be RequireObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *requestedLocales* be CanonicalizeLocaleList(*locales*).
5. ReturnIfAbrupt(*requestedLocales*).
6. Let *len* be the number of elements in *requestedLocales*.
7. If *len* > 0, then
   a. Let *requestedLocale* be the first element of *requestedLocales*.
8. Else
   a. Let *requestedLocale* be DefaultLocale().
9. Let *noExtensionsLocale* be the String value that is *requestedLocale* with all Unicode locale extension sequences (6.2.1) removed.
10. Let *availableLocales* be a List with the language tags of the languages for which the Unicode character database contains language sensitive case mappings.
11. Let *locale* be BestAvailableLocale(*availableLocales*, *noExtensionsLocale*).
12. If *locale* is **undefined**, let *locale* be `"und"`.
13. Let *cpList* be a List containing in order the code points of *S* as defined in ES2015, 6.1.4, starting at the first element of *S*.
14. For each code point *c* in *cpList*, if the Unicode Character Database provides a lower case equivalent of *c* that is either language insensitive or for the language *locale*, then replace *c* in *cpList* with that/those equivalent code point(s).
15. Let *cuList* be a new List.
16. For each code point *c* in *cpList*, in order, append to *cuList* the elements of the UTF-16 Encoding (defined in ES2015, 6.1.4) of *c*.
17. Let *L* be a String whose elements are, in order, the elements of *cuList*.
18. Return *L*.

The result must be derived according to the case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file, but also the SpecialCasings.txt file that accompanies it).

The value of the **`length`** property of the **`toLocaleLowerCase`** method is 0.

NOTE 1 As of Unicode 5.1, the *availableLocales* list contains the elements **`"az"`**, **`"lt"`**, and **`"tr"`**.

NOTE 2 The case mapping of some code points may produce multiple code points. In this case the result String may not be the same length as the source String. Because both **`toLocaleUpperCase`** and **`toLocaleLowerCase`** have context-sensitive behaviour, the functions are not symmetrical. In other words, **`s.toLocaleUpperCase().toLocaleLowerCase()`** is not necessarily equal to **`s.toLocaleLowerCase()`**.

NOTE 3 The **`toLocaleLowerCase`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

13.1.3 String.prototype.toLocaleUpperCase ([locales])

This definition supersedes the definition provided in ES2015, 21.1.3.21.

This function interprets a string value as a sequence of code points, as described in ES2015, 6.1.4. This function behaves in exactly the same way as **`String.prototype.toLocaleLowerCase`**, except that characters are mapped to their *uppercase* equivalents as specified in the Unicode character database.

The value of the **`length`** property of the **`toLocaleUpperCase`** method is 0.

NOTE The **`toLocaleUpperCase`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

13.2 Properties of the Number Prototype Object

The following definition(s) refer to the abstract operation thisNumberValue as defined in ES2015, 20.1.3.

13.2.1 Number.prototype.toLocaleString ([locales [, options]])

This definition supersedes the definition provided in ES2015, 20.1.3.4.

When the **`toLocaleString`** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be thisNumberValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. Let *numberFormat* be Construct(**`%NumberFormat%, «locales, options»)`**.
4. ReturnIfAbrupt(*numberFormat*).
5. Return FormatNumber(*numberFormat*, *x*).

The value of the **`length`** property of the **`toLocaleString`** method is **`0`**.

13.3 Properties of the Date Prototype Object

The following definition(s) refer to the abstract operation thisTimeValue as defined in ES2015, 20.3.4.

13.3.1 Date.prototype.toLocaleString ([locales [, options]])

This definition supersedes the definition provided in ES2015, 20.3.4.39.

When the **toLocaleString** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be thisTimeValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. If *x* is **NaN**, return **"Invalid Date"**.
4. Let *options* be ToDateTimeOptions(*options*, **"any"**, **"all"**).
5. ReturnIfAbrupt(*options*).
6. Let *dateFormat* be Construct(**%DateTimeFormat%,«locales, options»)**.
7. ReturnIfAbrupt(*dateFormat*).
8. Return FormatDateTime(*dateFormat*, *x*).

The value of the **length** property of the **toLocaleString** method is **0**.

13.3.2 Date.prototype.toLocaleDateString ([locales [, options]])

This definition supersedes the definition provided in ES2015, 20.3.4.38.

When the **toLocaleDateString** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be thisTimeValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. If *x* is **NaN**, return **"Invalid Date"**.
4. Let *options* be ToDateTimeOptions(*options*, **"date"**, **"date"**).
5. ReturnIfAbrupt(*options*).
6. Let *dateFormat* be Construct(**%DateTimeFormat%, «locales, options»)**.
7. ReturnIfAbrupt(*dateFormat*).
8. Return FormatDateTime(*dateFormat, x*).

The value of the **length** property of the **toLocaleDateString** method is **0**.

13.3.3 Date.prototype.toLocaleTimeString ([locales [, options]])

This definition supersedes the definition provided in ES2015, 20.3.4.40.

When the **toLocaleTimeString** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be thisTimeValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. If x is **NaN**, return **"Invalid Date"**.
4. Let *options* be ToDateTimeOptions(*options*, **"time"**, **"time"**).
5. ReturnIfAbrupt(*options*).
6. Let *timeFormat* be Construct(**%DateTimeFormat%, «locales, options»)**.
7. ReturnIfAbrupt(*timeFormat*),
8. Return FormatDateTime(*timeFormat, x*).

The value of the **length** property of the **toLocaleTimeString** method is **0**.

13.4 Properties of the Array Prototype Object

13.4.1 Array.prototype.toLocaleString([locales [, options]])

This definition supersedes the definition provided in ES2015, 22.1.3.26.

The following steps are taken:

1. Let *A* be ToObject(**this** value).
2. ReturnIfAbrupt(*A*).
3. Let *len* be ToLength(Get(*A*, **"length"**)).
4. ReturnIfAbrupt(*len*).
5. Let *separator* be the String value for the list-separator String appropriate for the host environment's current locale (this is derived in an implementation-defined way).
6. If *len* is zero, return the empty String.
7. Let *firstElement* be Get(*A*, **"0"**).
8. ReturnIfAbrupt(*firstElement*).
9. If *firstElement* is **undefined** or **null**, then
   a. Let *R* be an empty String.
10. Else
    a. Let *R* be ToString(Invoke(*firstElement*, **"toLocaleString", «locales, options»**)).
    b. ReturnIfAbrupt(*R*).
11. Let *k* be 1.
12. Repeat, while *k* < *len*
    a. Let *S* be a String value produced by concatenating *R* and *separator*.
    b. Let *nextElement* be Get(*A*, ToString(*k*)).
    c. ReturnIfAbrupt(*nextElement*).
    d. If *nextElement* is **undefined** or **null**, then
       i. Let *R* be the empty String.
    e. Else
       i. Let *R* be ToString(Invoke(*nextElement*, **"toLocaleString", «locales, options»**)).
       ii. ReturnIfAbrupt(*R*).
    f. Let *R* be a String value produced by concatenating *S* and *R*.
    g. Increase *k* by 1.
13. Return *R*.

NOTE 1 The elements of the array are converted to Strings using their **toLocaleString** methods, and these Strings are then concatenated, separated by occurrences of a separator String that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of **toString**, except that the result of this function is intended to be locale-specific.

NOTE 2 The **toLocaleString** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

Annex A
(informative)

Implementation Dependent Behaviour

The following aspects of the ECMAScript 2015 Internationalization API Specification are implementation dependent:

- In all functionality:
  - Additional values for some properties of *options* arguments (2)
  - Canonicalization of extension subtag sequences beyond the rules of RFC 5646 (6.2.3)
  - The default locale (6.2.4)
  - The default time zone (6.4.3)
  - The set of available locales for each constructor (9.1)
  - The BestFitMatcher algorithm (9.2.4)
  - The BestFitSupportedLocales algorithm (9.2.7)
- In Collator:
  - Support for the Unicode extensions keys kn, kf and the parallel options properties numeric, caseFirst (10.1.1)
  - The set of supported `"co"` key values (collations) per locale beyond a default collation (10.2.3)
  - The set of supported `"kn"` key values (numeric collation) per locale (10.2.3)
  - The set of supported `"kf"` key values (case order) per locale (10.2.3)
  - The default search sensitivity per locale (10.2.3)
  - The sort order for each supported locale and options combination (10.3.2)
- In NumberFormat:
  - The set of supported `"nu"` key values (numbering systems) per locale (11.2.3)
  - The patterns used for formatting positive and negative values as decimal, percent, or currency values per locale (11.3.2)
  - Localized representations of **NaN** and **Infinity** (11.3.2)
  - The implementation of numbering systems not listed in Table 2 (11.3.2)
  - Localized decimal and grouping separators (11.3.2)
  - Localized digit grouping schemata (11.3.2)
  - Localized currency symbols and names (11.3.2)
- In DateTimeFormat:
  - The BestFitFormatMatcher algorithm (12.1.1)
  - The set of supported `"ca"` key values (calendars) per locale (12.2.3)
  - The set of supported `"nu"` key values (numbering systems) per locale (12.2.3)
  - The default hour12 and hourNo0 settings per locale (12.2.3)
  - The set of supported date-time formats per locale beyond a core set, including the representations used for each component and the associated patterns (12.2.3)
  - Localized weekday names, era names, month names, am/pm indicators, and time zone names (12.3.2)
  - The calendric calculations used for calendars other than `"gregory"`, and adjustments for local time zones and daylight saving time (12.3.2)

Annex B
(informative)

Additions and Changes That Introduce Incompatibilities with Prior Editions

- 10.1, 11.1, 12.1 In ECMA-402, 1st Edition, constructors could be used to create Intl objects from arbitrary objects. This is no longer possible in 2nd Edition.